

type-driven program synthesis

Nadia Polikarpova



UCSDCSE
Computer Science and Engineering

my goal: automate programming



example: insert into a sorted list

Input:

x

5

xs

1	2	7	8
---	---	---	---

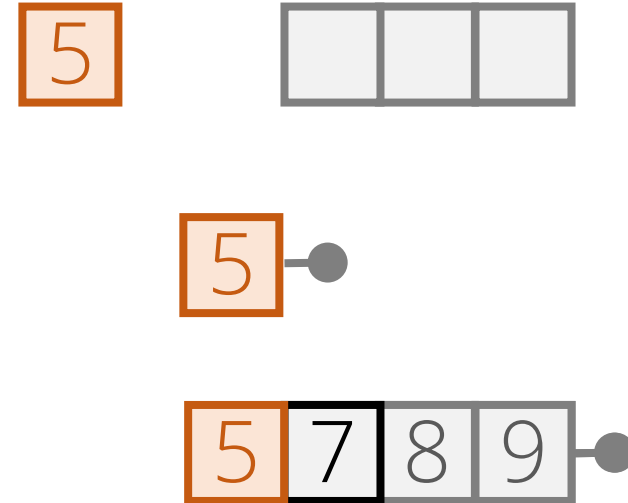
Output:

ys

1	2	5	7	8
---	---	---	---	---

insert in a functional language

```
insert x xs =  
  match xs with  
  Nil →  
    Cons x Nil  
  Cons h t →  
    if x ≤ h  
    then Cons x xs  
    else Cons h (insert x t)
```



```

void insert(node *xs, int x) {
    node *new;
    node *temp;
    node *prev;

    new = (node *)malloc(sizeof(node));
    if(new == NULL) {
        printf("Insufficient memory.");
        return;
    }
    new->val = x;
    new->next = NULL;
    if (xs == NULL) {
        xs = new;
    } else if(x < xs->val) {
        new->next = xs;
        xs = new;
    } else {
        prev = xs;
        temp = xs->next;
        while(temp != NULL && x > temp->val) {
            prev = temp;
            temp = temp->next;
        }
        if(temp == NULL) {
            prev->next = new;
        } else {
            new->next = temp;
            prev->next = new;
        }
    }
}

```

C

automates
pointer manipulation &
memory management



```

insert x xs =
  match xs with
    Nil →
      Cons x Nil
    Cons h t →
      if x ≤ h
      then Cons x xs
      else Cons h (insert x t)

```

Haskell

what's next?

```
append:
    push ebp
    mov ebp, esp
    push eax
    push ebx
    push len
    call malloc
    mov ebx, [ebp + 12]
    mov [eax + info], ebx
    mov dword [eax + next], 0
    mov ebx, [ebp + 8]
    cmp dword [ebx], 0
    je null_pointer
    mov ebx, [ebx]

next_element:
    cmp dword [ebx + next], 0
    je found_last
    mov ebx, [ebx + next]
    jmp next_element

found_last:
    push eax
    push addMes
    call puts
    add esp, 4
    pop eax
    mov [ebx + next], eax

go_out:
    pop ebx
    pop eax
    mov esp, ebp
    pop ebp
    ret 8

null_pointer:
    push eax
    push nullMes
    call puts
    add esp, 4
    pop eax
    mov [ebx], eax
    jmp go_out
```

Assembly

```
void insert(node *xs, int x) {
    node *new;
    node *temp;
    node *prev;

    new = (node *)malloc(sizeof(node));
    if(new == NULL) {
        printf("Insufficient memory.");
        return;
    }
    new->val = x;
    new->next = NULL;
    if (xs == NULL) {
        xs = new;
    } else if(x < xs->val) {
        new->next = xs;
        xs = new;
    } else {
        prev = xs;
        temp = xs->next;
        while(temp != NULL && x > temp->val) {
            prev = temp;
            temp = temp->next;
        }
        if(temp == NULL) {
            prev->next = new;
        } else {
            new->next = temp;
            prev->next = new;
        }
    }
}
```

C

← automates
argument passing
& memory access

```
insert x xs =
  match xs with
  Nil → Cons x Nil
  Cons h t →
    if x ≤ h
    then Cons x xs
    else Cons h (insert x t)
```

Haskell

?

future →

this talk

I. specifications

are the new cool language feature

II. program synthesis

is the way to make it happen

III. synthesis-aided programming

is the the future

this talk

I. specifications

II. program synthesis

III. synthesis-aided programming

this talk

I. specifications

how do I tell the machine what I want?

specification for insert

Input:

x

xs : sorted list

Output:

ys : sorted list

$\text{elems } ys = \text{elems } xs \cup \{x\}$

easy to write 😊

hard to execute 😞

program synthesis

specification

in: x , sorted xs
out: sorted ys
 $\text{elems } ys = \text{elems } xs \cup \{x\}$



code

```
match xs with  
Nil → Cons x Nil  
Cons h t →  
  if x ≤ h  
  then Cons x xs  
  else Cons h (insert x t)
```

program synthesis

specification

in: x , sorted xs
out: sorted ys
 $\text{elems } ys = \text{elems } xs \cup \{x\}$



Synquid



code

```
match xs with
Nil → Cons x Nil
Cons h t →
  if x ≤ h
  then Cons x xs
  else Cons h (insert x t)
```

program synthesis

specification

in: x , sorted xs
out: sorted ys
 $\text{elems } ys = \text{elems } xs \cup \{x\}$

formalize this



Synquid



code

```
match xs with
Nil → Cons x Nil
Cons h t →
  if x ≤ h
  then Cons x xs
  else Cons h (insert x t)
```

types are specifications

```
insert :: a → List a → List a
```

types are specifications

$\text{Ord } a \Rightarrow$
`insert` :: `a` $\rightarrow$ `List a` $\rightarrow$ `List a`

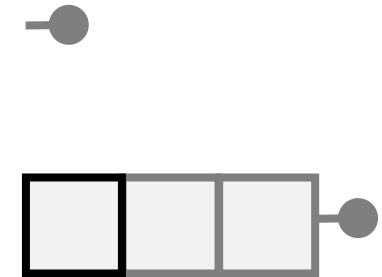
ordinary types are insufficient!

refinement types

$$\{ \underbrace{v:\text{Int}}_{\text{shape}} \mid \underbrace{0 \leq v}_{\text{refinement}} \}$$

from lists to sorted lists

```
data List a where
  Nil :: List a
  Cons :: h:a →
         t:List a →
         List a
```



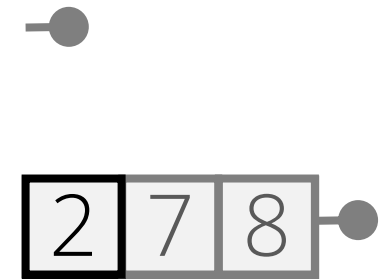
from lists to sorted lists

```
data SList a where
```

```
Nil :: SList a
```

```
Cons :: h:a →
```

```
  t:SList {v:a | h ≤ v} →  
  SList a
```



all you need
is one simple predicate!

insert in Synquid

```
insert :: x:a → xs:SList a →  
        {v:SList a | elems v = elems xs U {x}}
```

Synquid generates correct
code in under a second!

this talk

I. specifications

II. program synthesis

III. synthesis-aided programming

this talk

II. program synthesis

how do we get from specification to code?

type-driven program synthesis

specification

```
insert :: x:a →  
xs:SList a →  
{v:SList a | elems v =  
  elems xs U {x}}
```



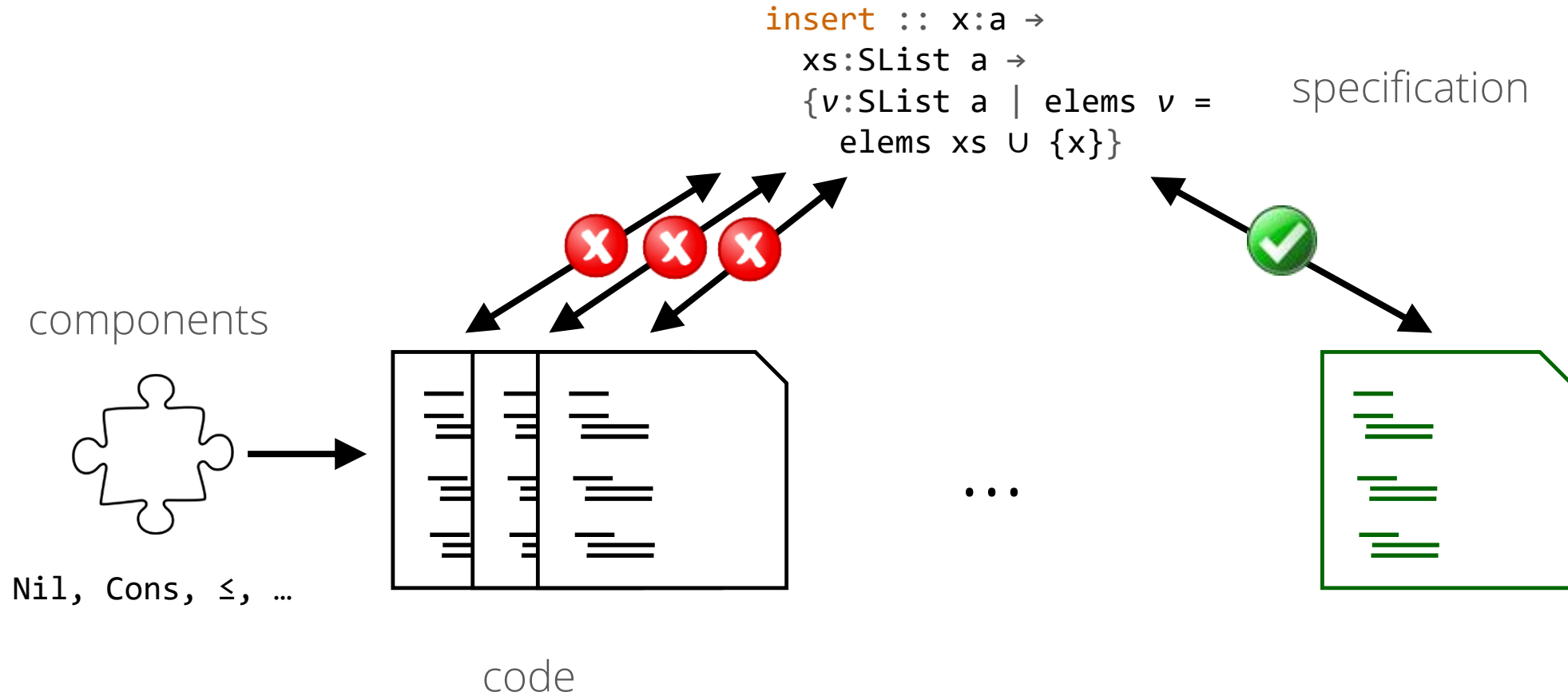
?



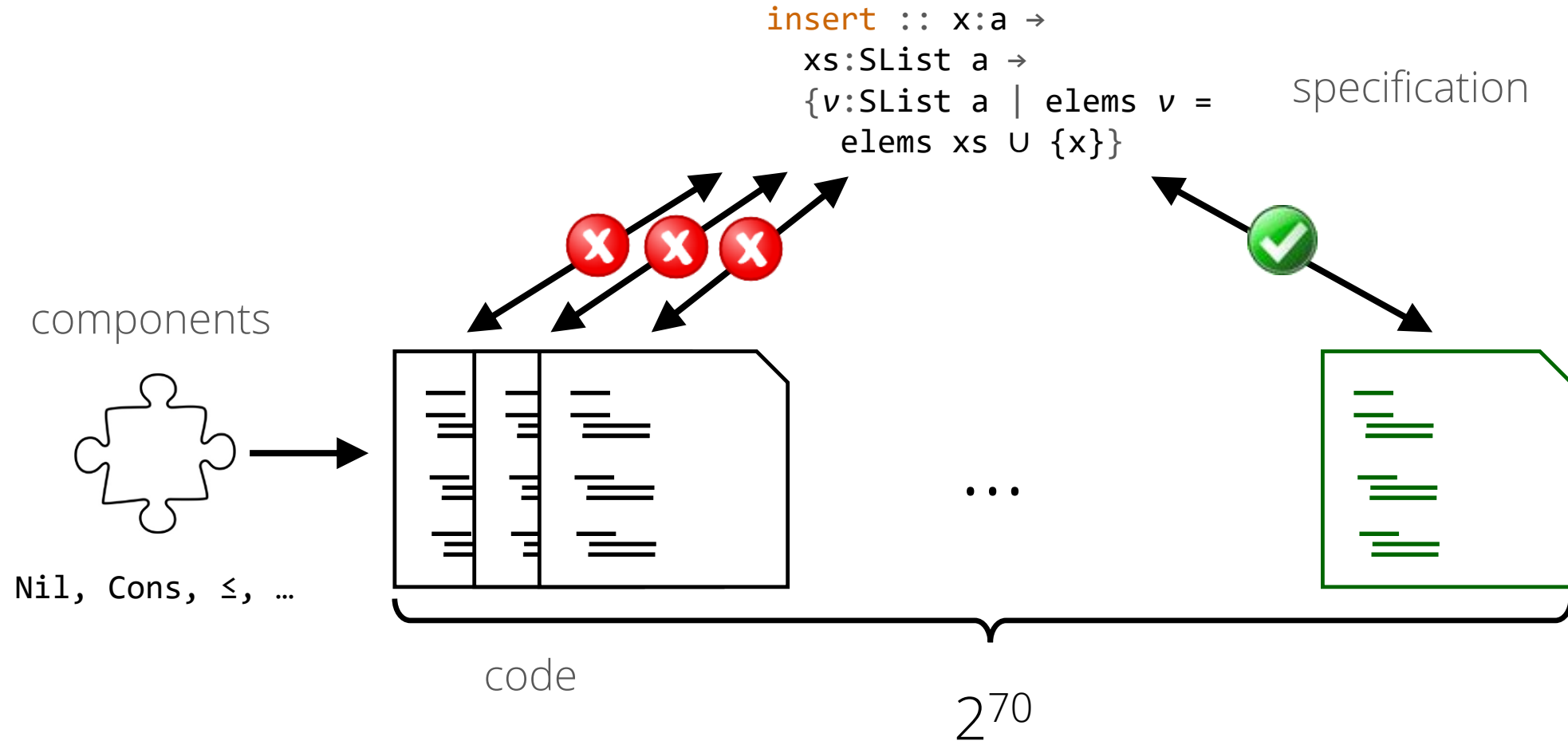
code

```
match xs with  
Nil → Cons x Nil  
Cons h t →  
  if x ≤ h  
  then Cons x xs  
  else Cons h (insert x t)
```

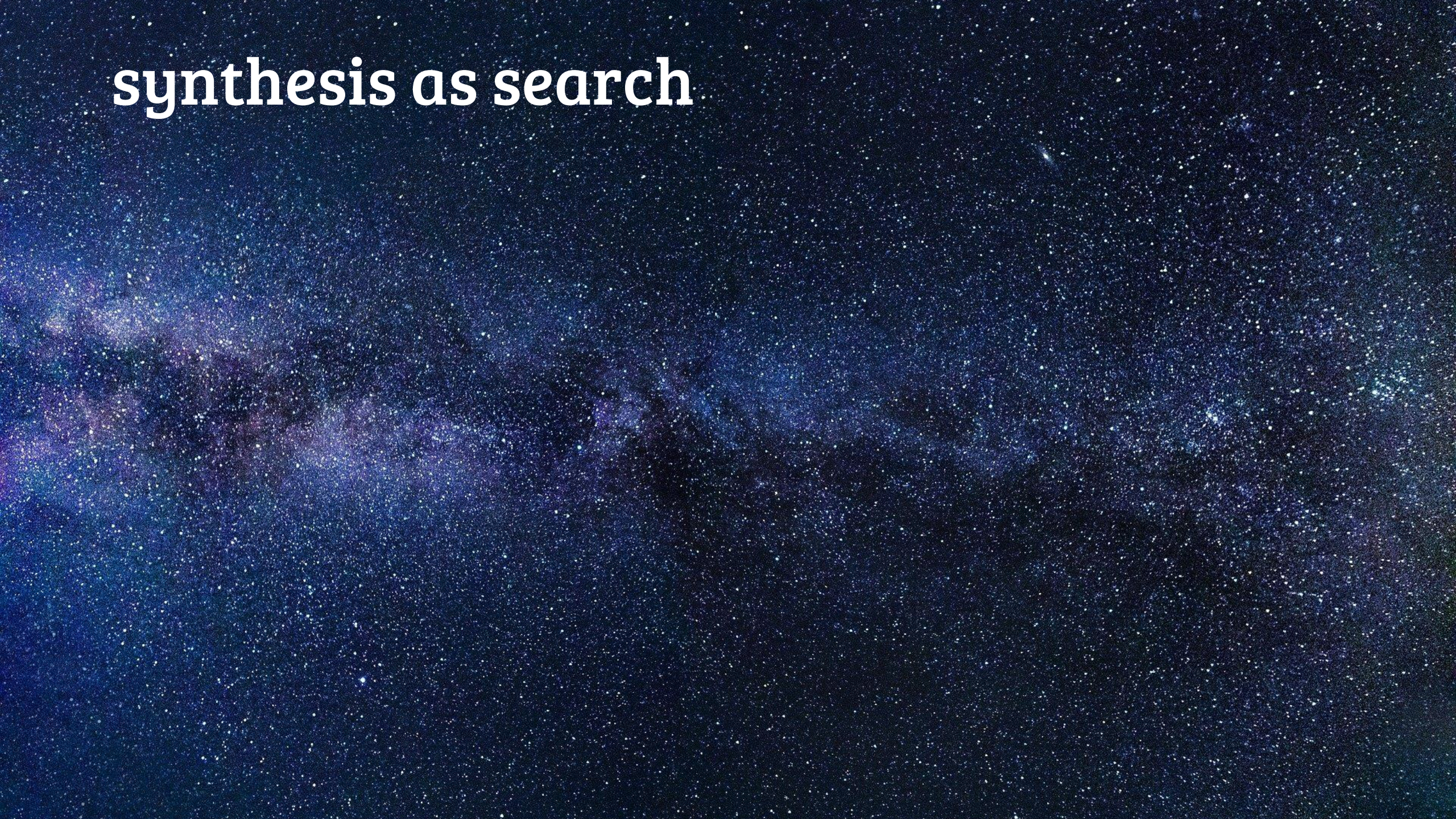
synthesis as search



synthesis as search



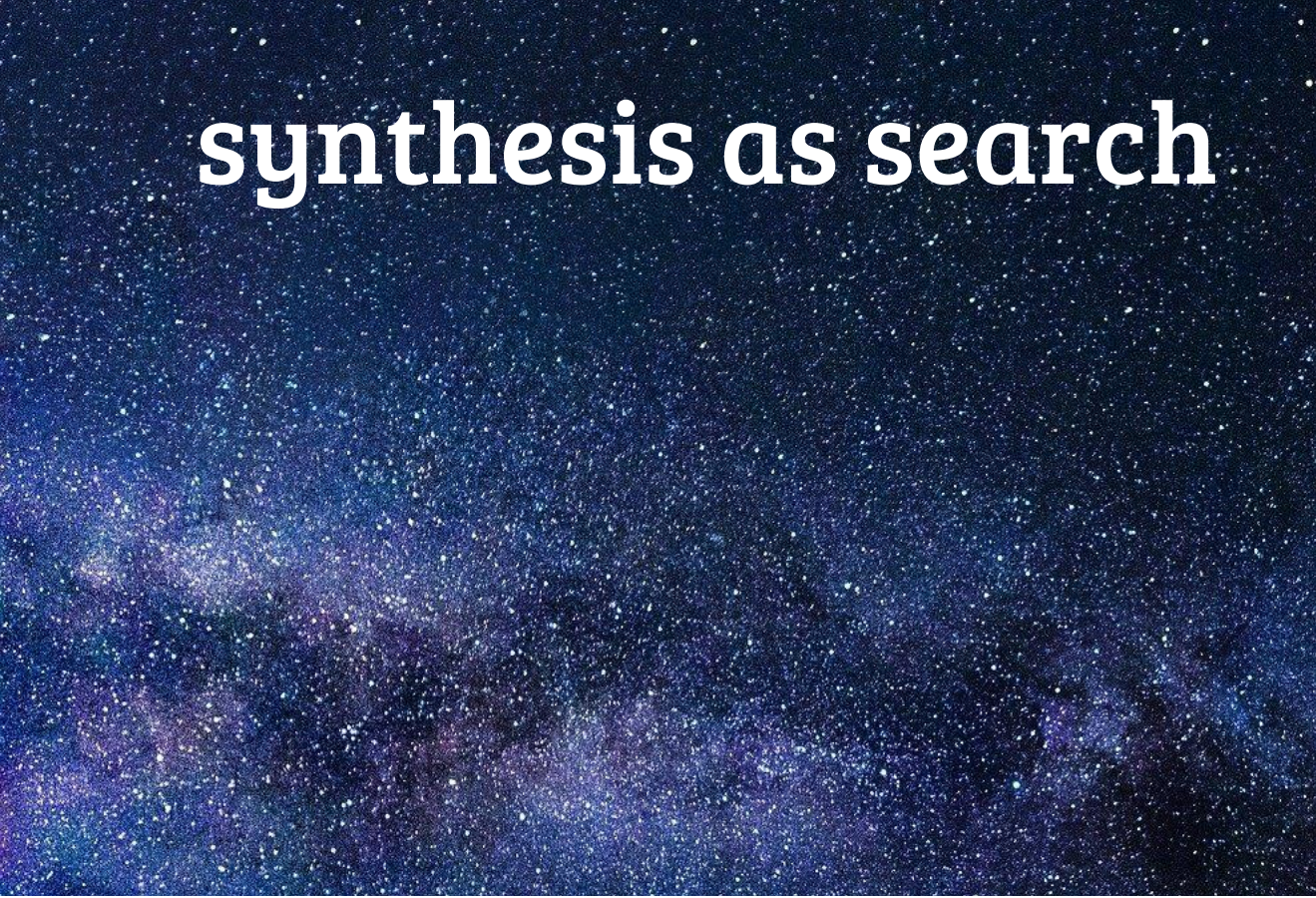
synthesis as search

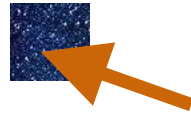


synthesis as search



synthesis as search



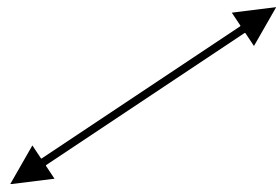
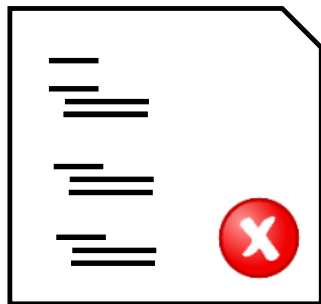


key idea: reject hopeless programs early

reject hopeless programs early

```
insert :: x:a →  
xs:SList a →  
{v:SList a | elems v =  
  elems xs U {x}}
```

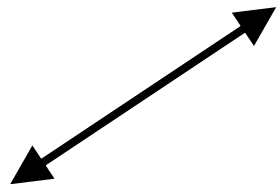
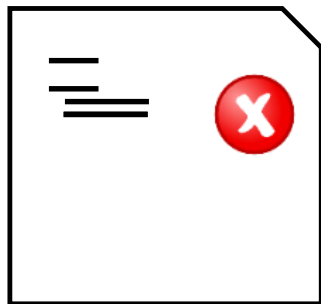
specification



reject hopeless programs early

```
insert :: x:a →  
xs:SList a →  
{v:SList a | elems v =  
  elems xs ∪ {x}}
```

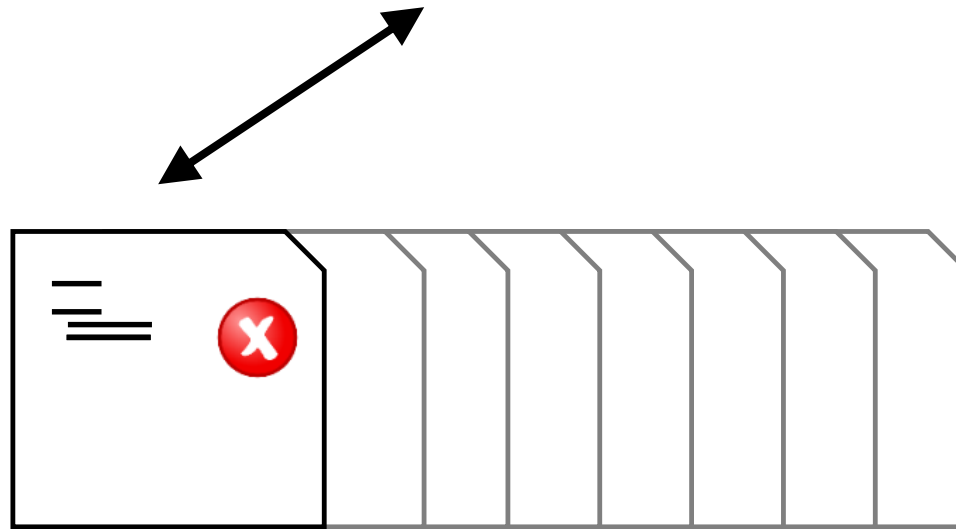
specification



reject hopeless programs early

```
insert :: x:a →  
xs:SList a →  
{v:SList a | elems v =  
  elems xs ∪ {x}}
```

specification



reject hopeless programs early

```
x:a → xs:SList a →  
  {v:SList a | elems v = elems xs ∪ {x}}
```



```
insert x xs =  
  match xs with  
    Nil → Nil  
    Cons h t → ??
```

reject hopeless programs early

```
x:a → xs:SList a →  
  {v:SList a | elems v = elems xs ∪ {x}}
```



```
insert x xs =  
  match xs with  
  Nil → Nil  
  Cons h t →
```



2^{50}

reject hopeless programs early

$\{v:\text{SList } a \mid \text{elems } v = \text{elems } xs \cup \{x\}\}$



```
insert x xs =  
  match xs with  
    Nil → Nil  
    Cons h t → ??
```

reject hopeless programs early

```
{v:SList a | elems v = {x}}
```



```
insert x xs =  
  match xs with  
  Nil → Nil  
  Cons h t → ??
```

reject hopeless programs early

`{v:SList a | elems v = {x}}`



`insert x xs =
 match xs with
 Nil → Nil
 Cons h t → ??`



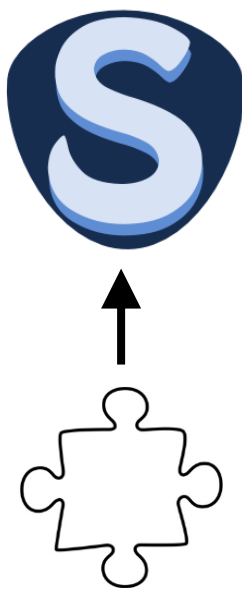
Expected
`{v:SList a | elems v = {x}}`
and got
`{v:SList a | elems v = {}}`

synquid

specification

```
insert :: x:a →  
xs:SList a →  
{v:SList a | elems v =  
  elems xs ∪ {x}}
```

components

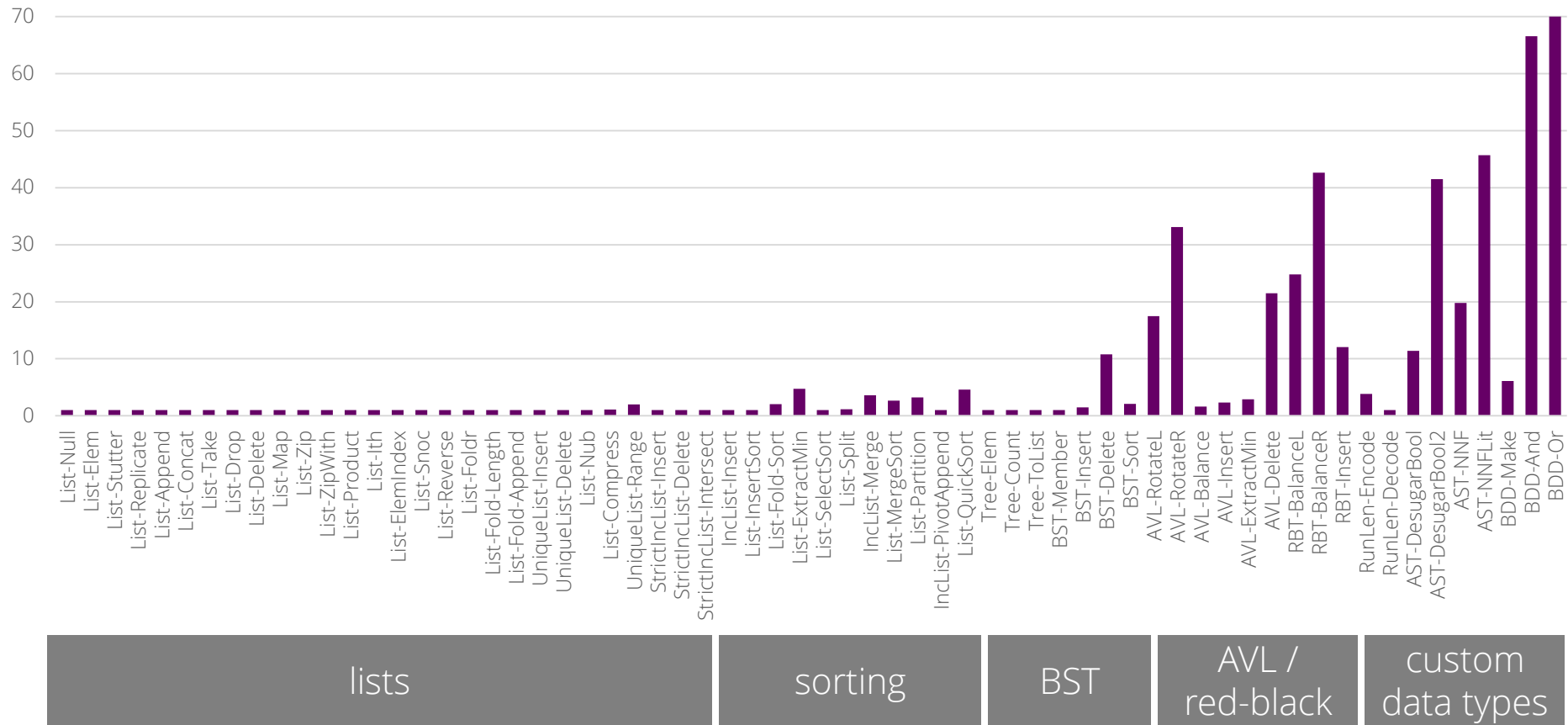


code

```
match xs with  
Nil → Cons x Nil  
Cons h t →  
  if x ≤ h  
  then Cons x xs  
  else Cons h (insert x t)
```

experiments

sec



case study: negation normal form

$$\neg(a \Rightarrow b \vee c)$$

1. only $\neg$, $\wedge$, $\vee$
2. $\neg$ only at variables

case study: negation normal form

$$\neg(a \Rightarrow b \vee c)$$

$\Rightarrow$ def.

$$\neg(\neg a \vee (b \vee c))$$

De Morgan

$$\neg\neg a \wedge \neg(b \vee c)$$

double-neg

$$a \wedge \neg(b \vee c)$$

De Morgan

$$a \wedge (\neg b \vee \neg c)$$

1. only $\neg$, $\wedge$, $\vee$

2. $\neg$ only at variables

nnf: data types

data Fml where

```
Var  :: String    → Fml
Not  :: Fml        → Fml
And  :: Fml → Fml → Fml
Or   :: Fml → Fml → Fml
Imp  :: Fml → Fml → Fml
```

data NNF where

```
NAtom :: String → Bool
      → NNF
NAnd  :: NNF → NNF → NNF
NOr   :: NNF → NNF → NNF
```

negated?



nnf: specification

data Fml **where**

```
Var  :: String    → Fml
Not  :: Fml       → Fml
And  :: Fml → Fml → Fml
Or   :: Fml → Fml → Fml
Imp  :: Fml → Fml → Fml
```

measure eval :: Fml → Bool **where**

```
Var v    → env v
Not f     → !(eval f)
And l r   → eval l && eval r
Or l r    → eval l || eval r
Imp l r   → eval l ==> eval r
```

data NNF **where**

```
NAtom :: String → Bool
      → NNF
NAnd  :: NNF → NNF → NNF
NOr   :: NNF → NNF → NNF
```

measure nEval :: NNF → Bool **where**

```
NAtom neg v → if neg then env v
              else !(env v)
NAnd l r    → nEval l && nEval r
NOr l r     → nEval l || nEval r
```

nnf: specification

`nnf :: f:Fml → {v:NNF | nEval v = eval f}`

nnf: synthesized code

```
nnf :: f:Fml → {v:NNF | nEval v = eval f}
nnf p = match p with
  BoolLiteral x2 → if x2
    then NOr (NAtom dummy x2) (NAtom dummy False)
    else NAnd (NAtom dummy x2) (NAtom dummy True)
  Var x16 → NAtom x16 False
  Not x20 → match x20 with
    BoolLiteral x22 → if x22
      then nnf (BoolLiteral False)
      else nnf (BoolLiteral True)
    Var x28 → NAtom x28 True
    Not x32 → nnf x32
    And x36 x37 → NOr (nnf (Not x36)) (nnf (Not x37))
    Or x46 x47 → NAnd (nnf (Not x46)) (nnf (Not x47))
    Implies x56 x57 → NAnd (nnf x56) (nnf (Not x57))
  And x65 x66 → NAnd (nnf x65) (nnf x66)
  Or x73 x74 → NOr (nnf x73) (nnf x74)
  Imp x81 x82 → NOr (nnf x82) (nnf (Not x81))
```

double-neg

De Morgan

⇒ def.

this talk

I. specifications

II. program synthesis

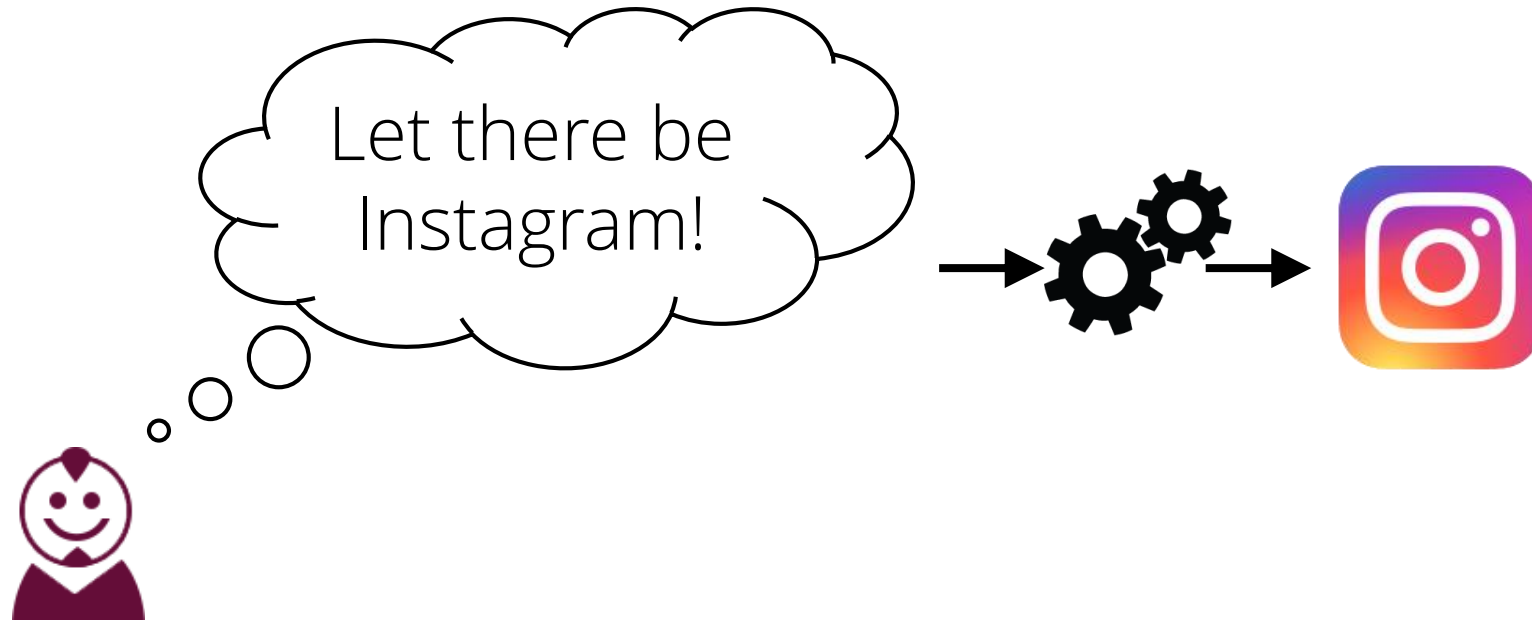
III. synthesis-aided programming

this talk

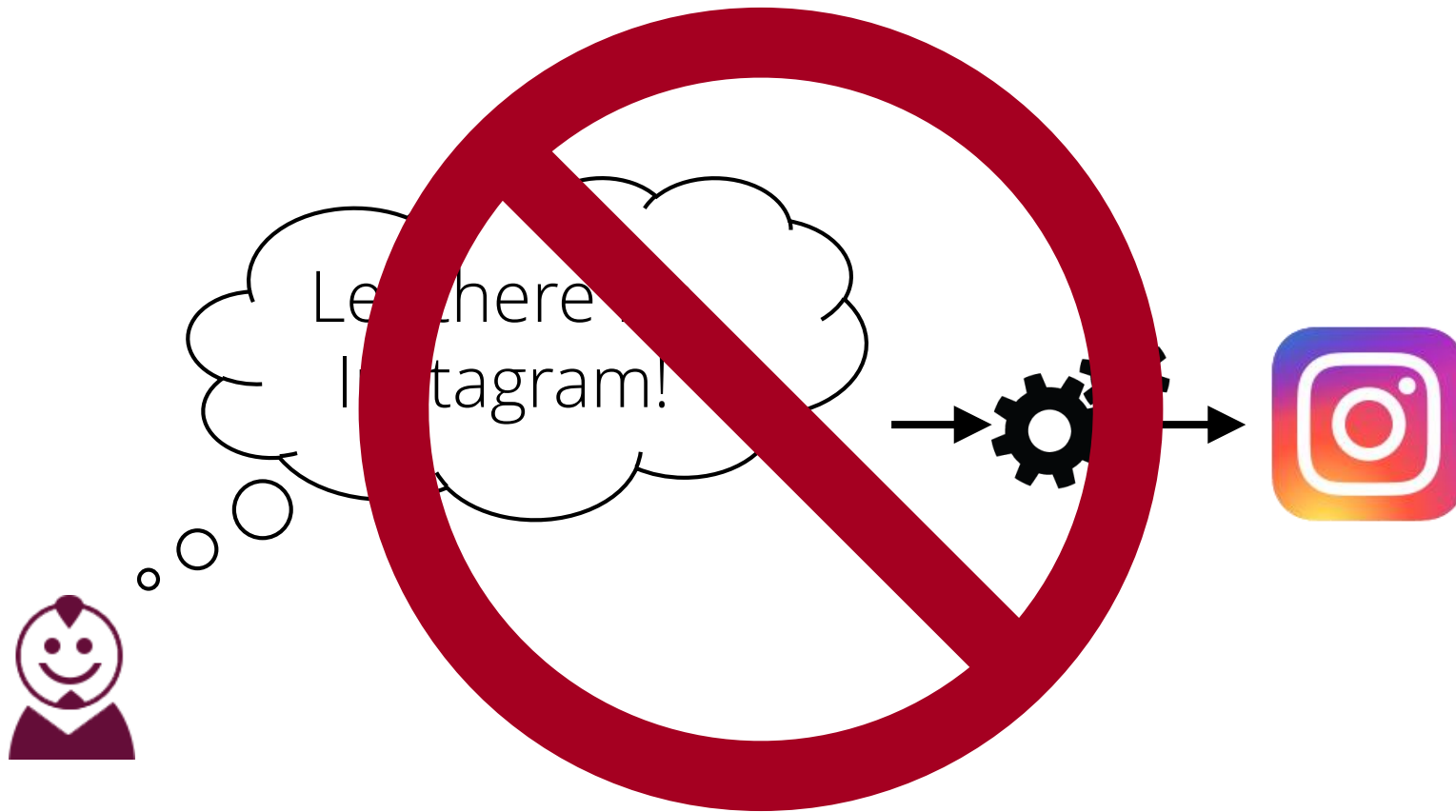
how can synthesis help us build software?

III. synthesis-aided programming

the future of programming

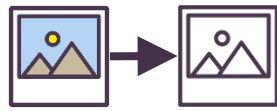


the future of programming **(not)**



orthogonal concerns

functional
requirements



data
invariants



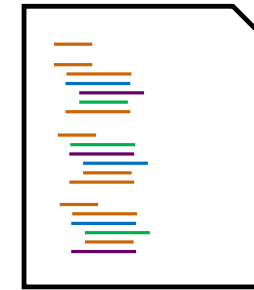
security
policies



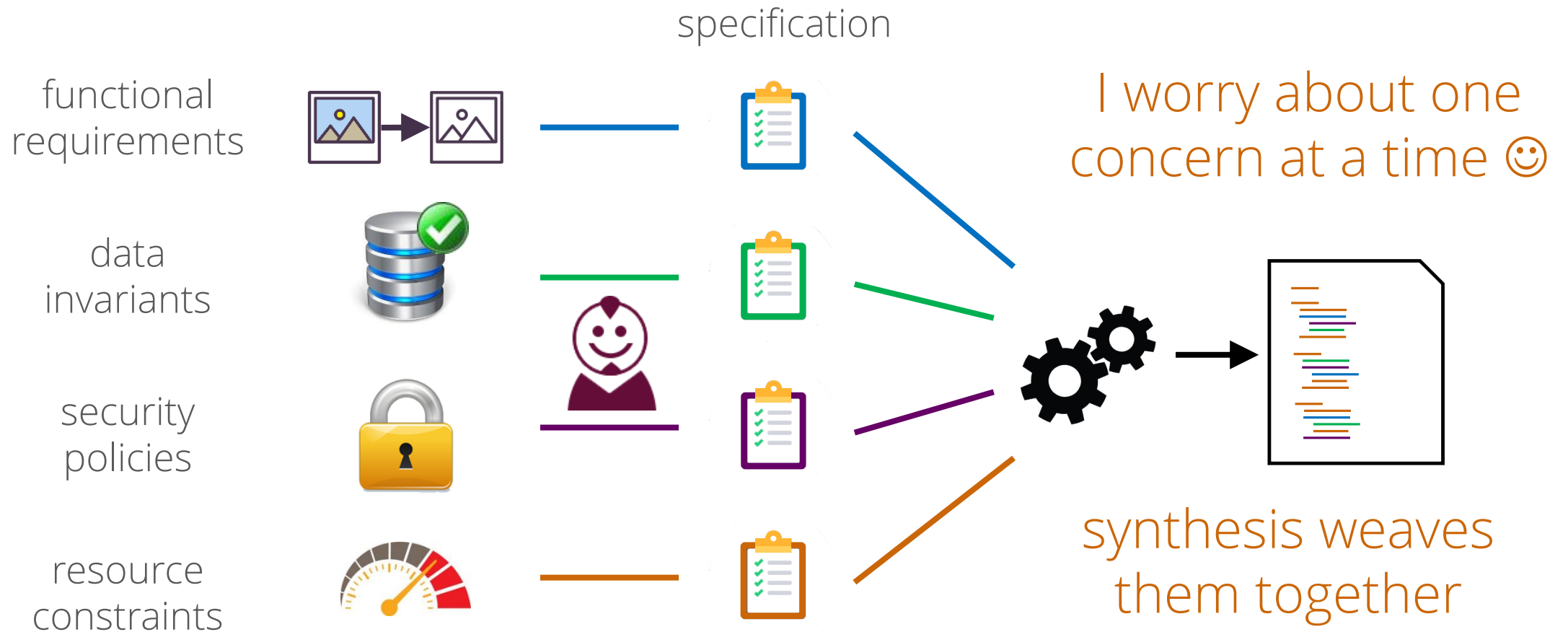
resource
constraints



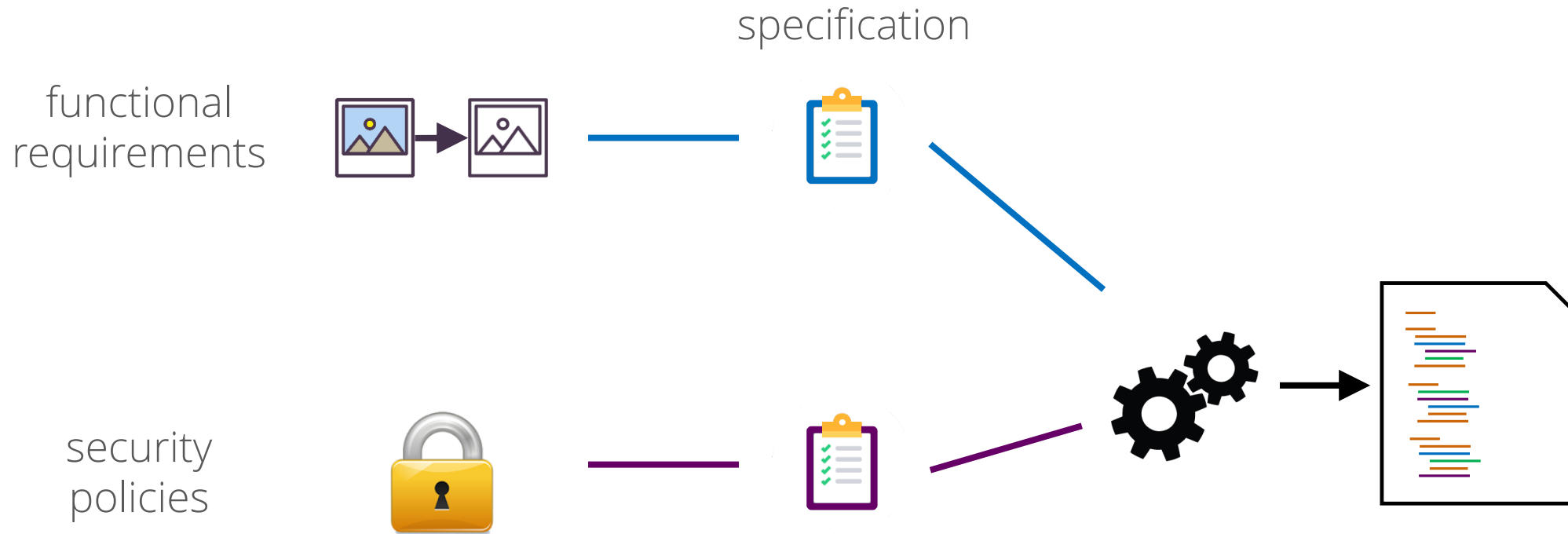
I worry about all
concerns together 😞



synthesis-aided programming



synthesis-aided programming



information leaks

electronic health records



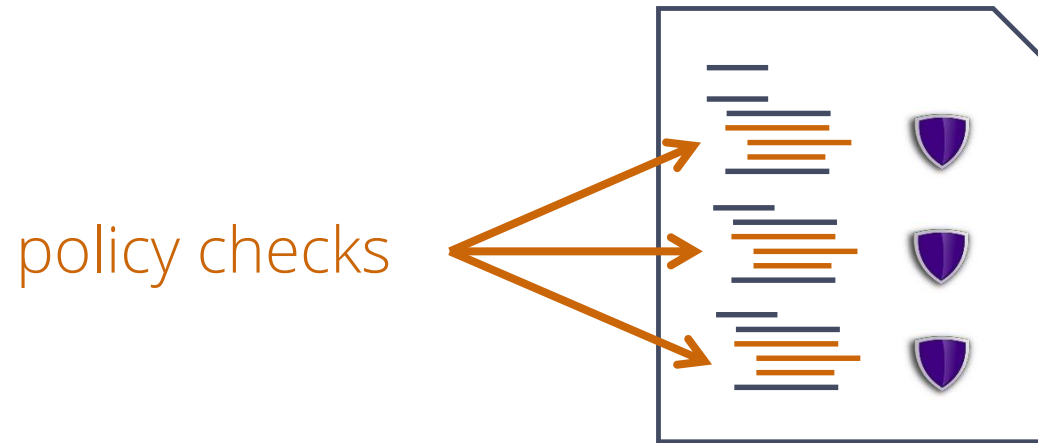
social networks



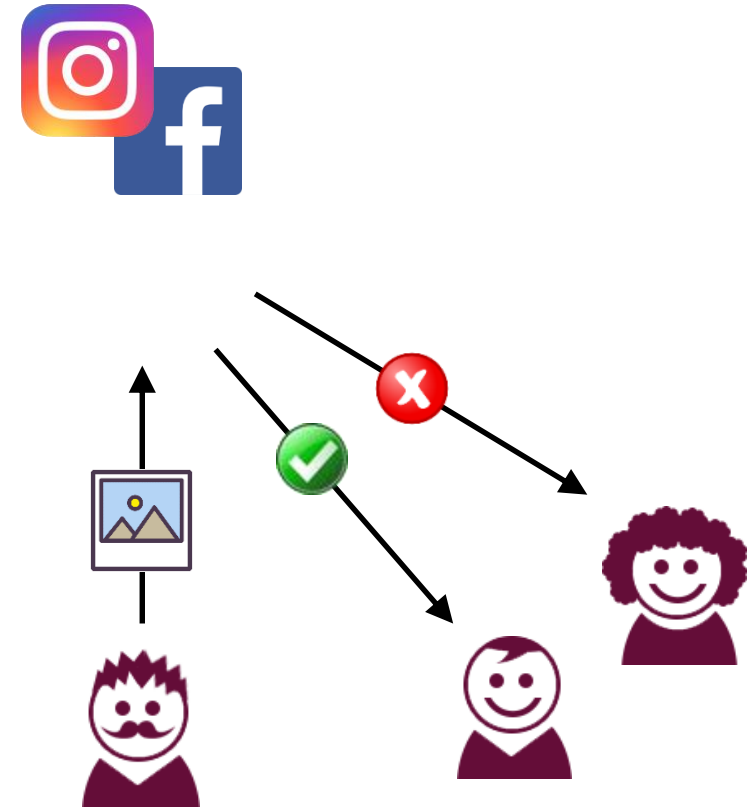
e-commerce



information leaks



social networks



information leaks

PRIVACY AND SECURITY

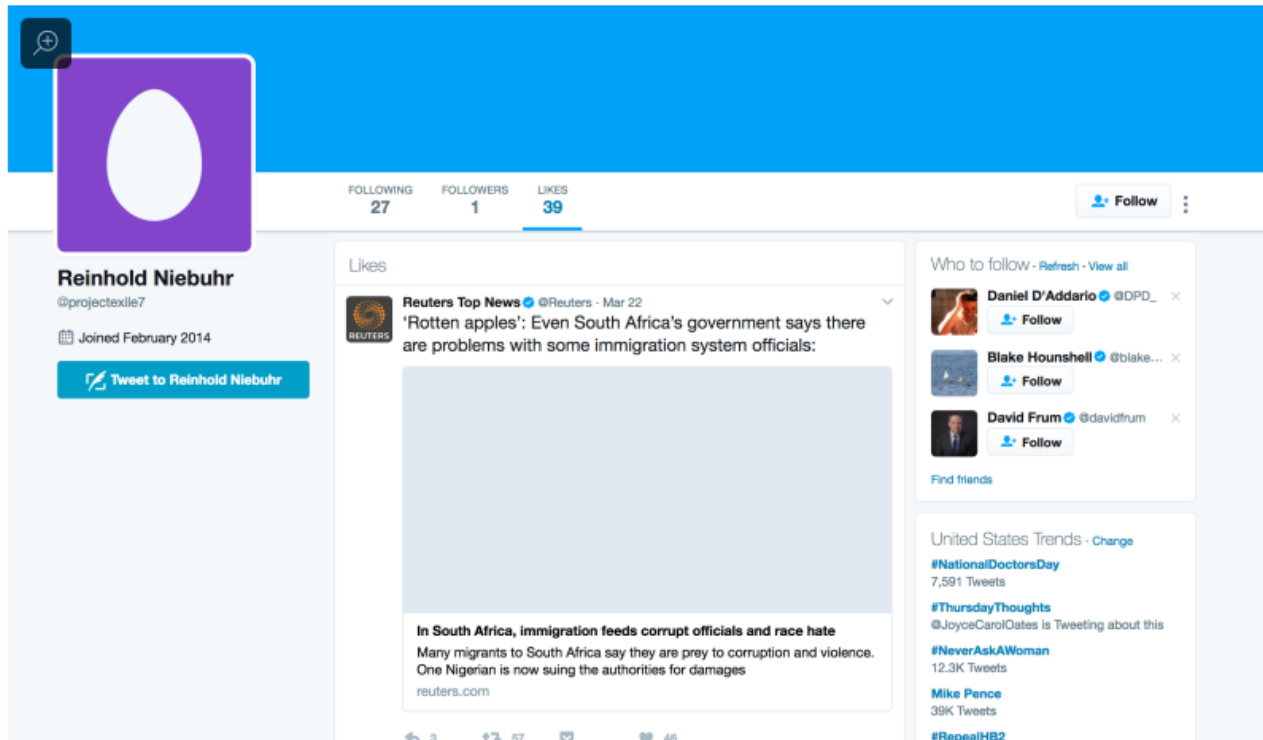
This Is Almost Certainly James Comey's Twitter Account



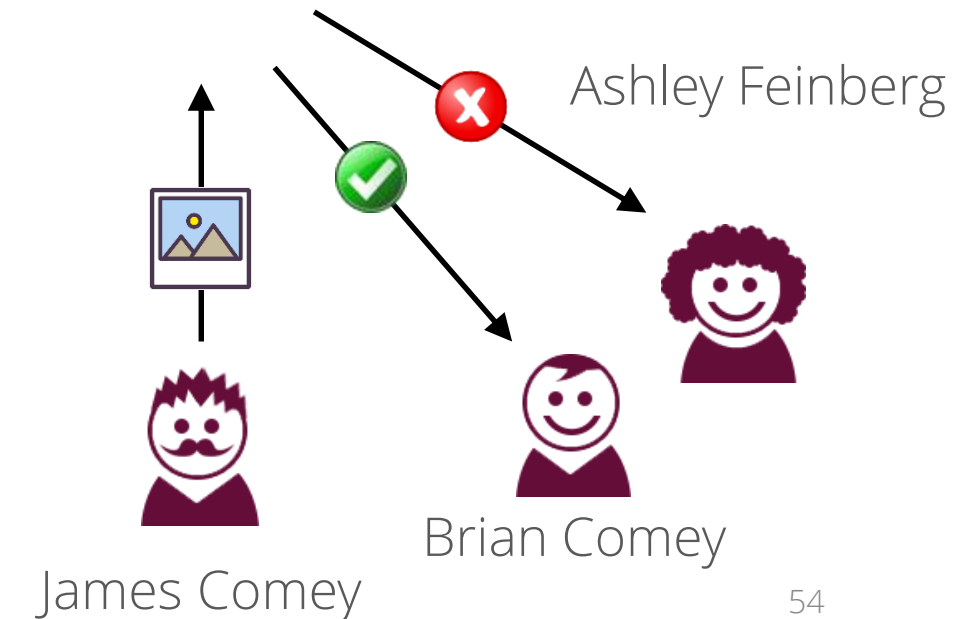
Ashley Feinberg

3/30/17 3:29pm • Filed to: JAMES COMEY ▾

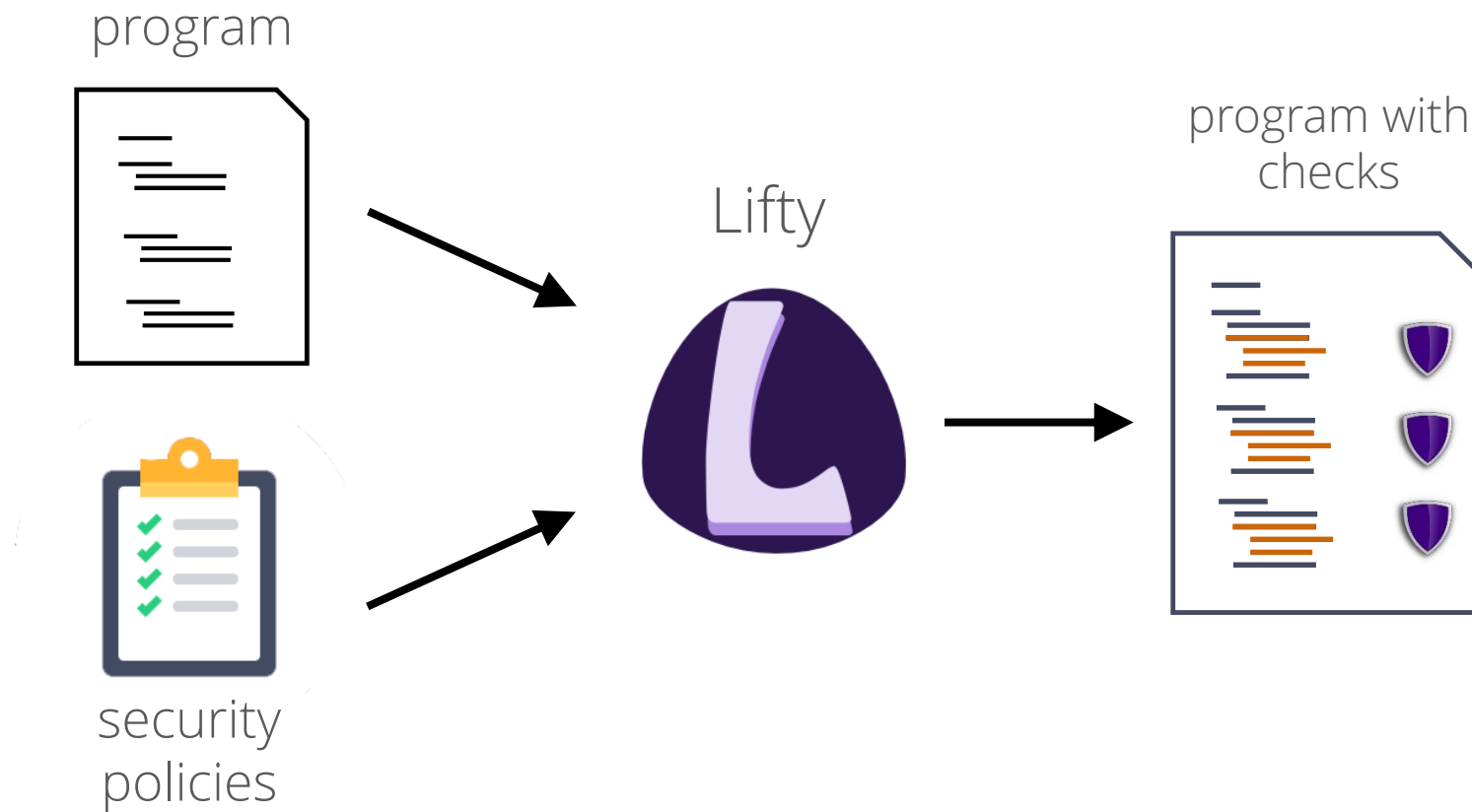
3.0M 672 238



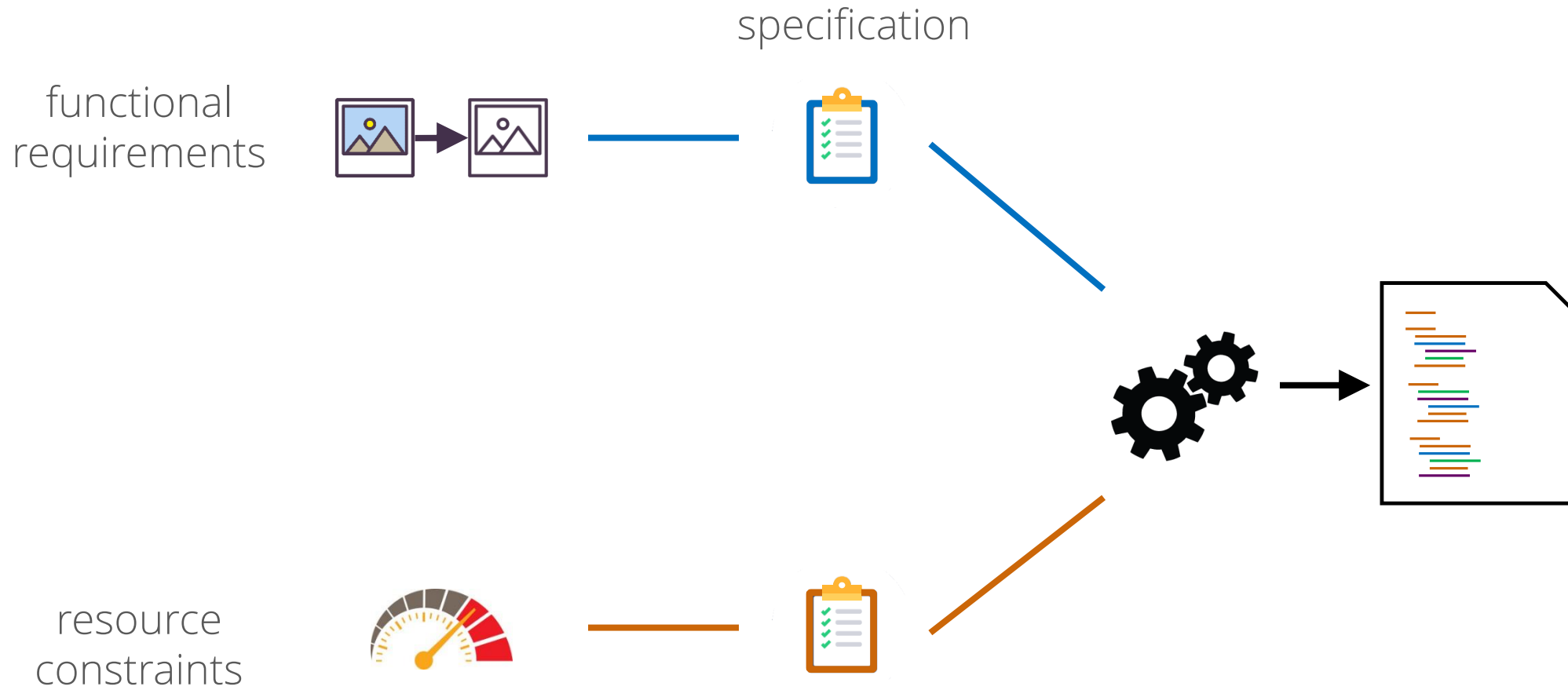
social networks



information security with Lifty

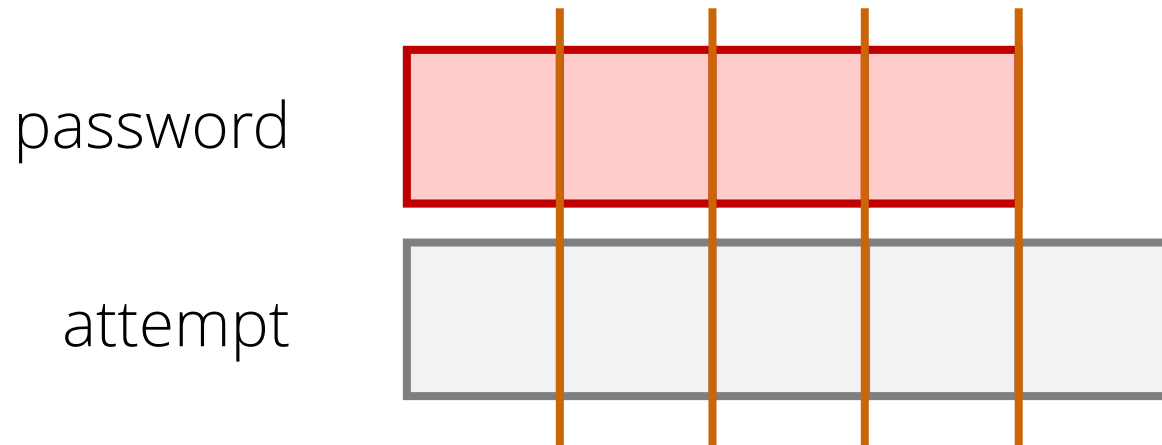


synthesis-aided programming



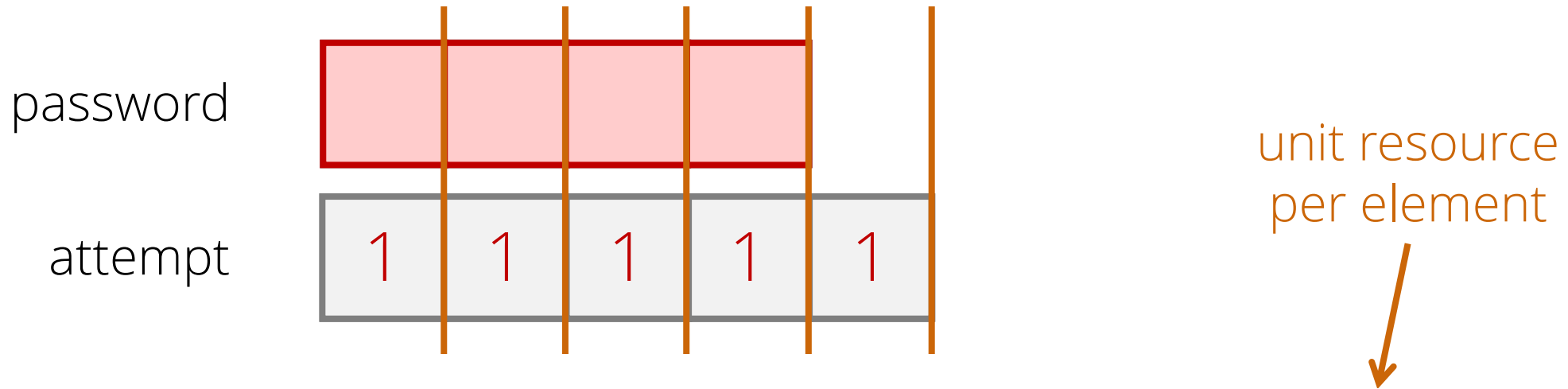
timing attacks

Compare the lengths of two strings:



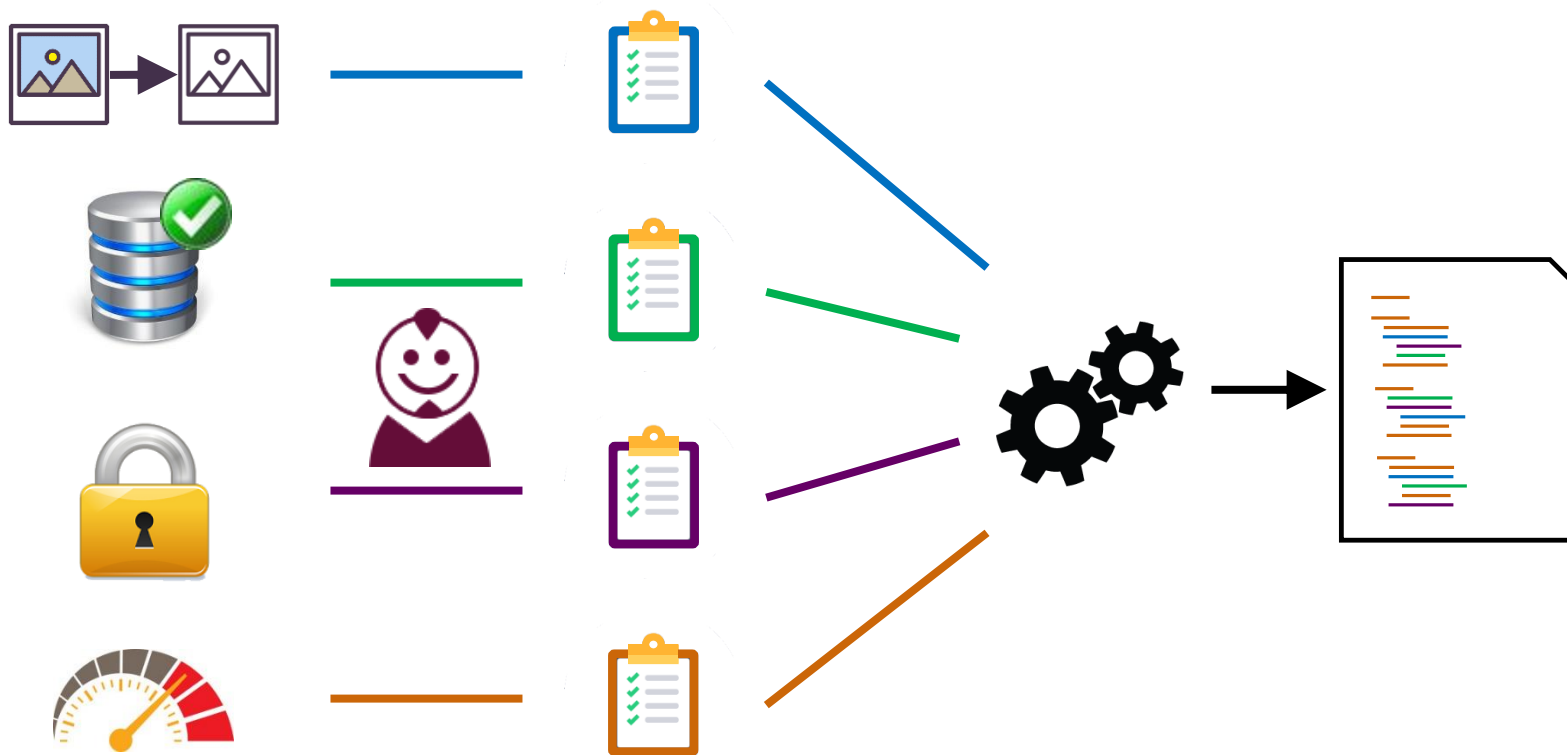
execution time
reveals length!

resource-aware programming with ReSyn



compare :: pass:List a → att:List {a || 1} →
 {v:Bool | v = (len key = len guess)}

synthesis-aided programming





<http://comcom.csail.mit.edu>